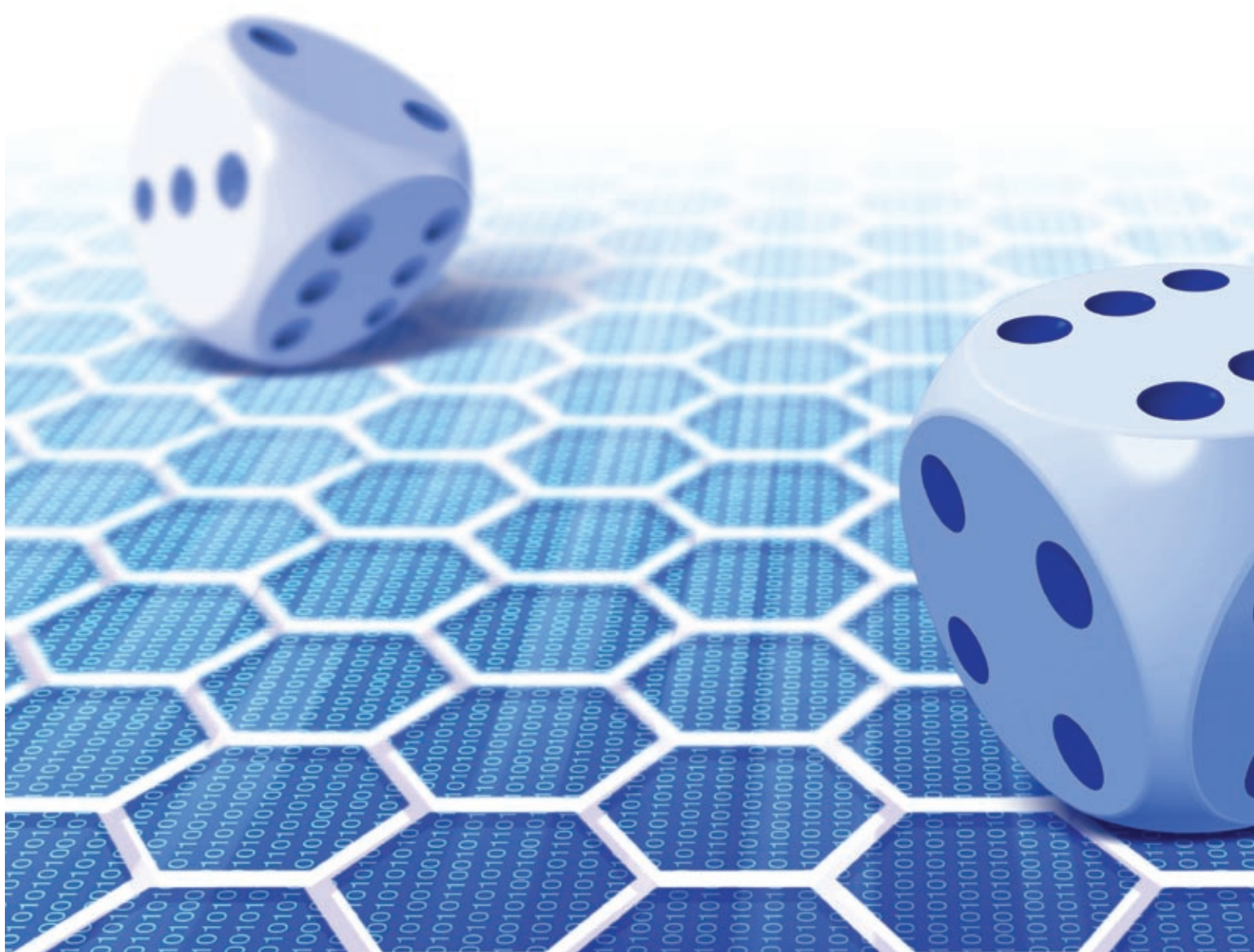




Computing the Uncomputable

**Do all problems have solutions?
Mike Bedford delves into the
problematic realm of logic –
and wonders if even the
seemingly unfathomable could
be figured out if we only had the
right computer**

Fans of Douglas Adams' spoof science fiction novel *The Hitchhiker's Guide to the Galaxy* will be familiar with Deep Thought, the supercomputer that was built to determine the answer to life, the universe, and everything. If you haven't read the book, let's just say that the answer turned out to be 42.

This might be a work of fiction, but it will come as no surprise that computers aren't suited to answering such questions of philosophy, and certainly not one so ill defined. But surely if a problem can be expressed in the language of logic or mathematics, it should be possible to write a computer program to solve it? If such a problem proves tricky – as speech recognition has, for instance – we might conclude that it will eventually

be cracked as PCs become faster and techniques improve. But this isn't always true. Mathematicians have felt for decades that some problems can never be solved, no matter how fast the computer and how clever the programmer. These problems have been described as uncomputable.

That's the traditional view, but some computer scientists are now starting to think the unthinkable: that it is possible to compute the uncomputable. But it would require a radically different type of computer called a hypercomputer. And when we say radically different, we mean just that. So far every computer that has ever existed – and this includes state-of-the-art technologies such as quantum computers and neural networks – have been incapable of solving uncomputable problems.



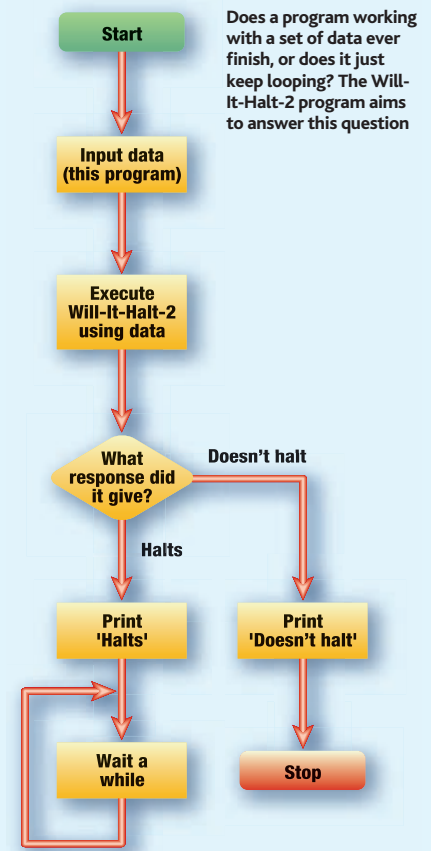
AN UNCOMPUTABLE PROBLEM

The halting problem is the first known uncomputable problem. It asks if a computer program working with a particular set of data will ever halt, or whether it will loop forever. With some programs, such as those that are just a loop with no exit from that loop, it would be easy to spot that it will never terminate. But how do we know that there is no general method that would work for any program? Here we look at the proof.

We start by assuming there is a program that solves the halting problem – let's call it Will-It-Halt. It requires two inputs, the program and the data, and it terminates giving one of two answers: yes or no. Let's use this program as a subroutine to a more complicated program, Will-It-Halt-2, that takes a single input, which is a program. It executes Will-It-Halt using its own input (a program) as both the inputs to Will-It-Halt. This might seem an odd thing to do since Will-It-Halt is expecting a program and some data as its inputs, but since an executable program is just a load of numbers it's perfectly valid to use it as data.

Now we create a more complicated program using Will-It-Halt-2 as a subroutine. The program is shown here as a flow diagram. Follow the flowchart to see what happens if we run this program using the program itself as its data. Either it says yes, which means it will halt but in that case it goes into an infinite loop so it doesn't halt. Or it says no, indicating that it won't halt, but then it does halt. In either case, it has given the wrong answer and the only explanation to this paradox is that we made an invalid assumption when we stated that there was a program, Will-It-Halt, that can solve the halting problem.

A Paradoxical Program



In this article, we investigate whether there really are questions that will never be answered, or if hypercomputers could provide the answers.

UNDERSTANDING THE UNCOMPUTABLE

There is one uncomputable problem that has existed for many hundreds of years. It is a simple problem, and one that is easy to understand, making it an ideal introduction to the concept.

In the church of Santa Maria in Cosmedin in Rome, tourists flock to see a gnarled stone face. This is the Bocca della Verità, the Mouth of Truth. According to legend, this sculpture is a universal oracle: it's able to tell whether any statement is true or false. You stand in front of the sculpture, put your hand in its open mouth and make a statement. If the statement is true, nothing happens. If it's false the mouth closes, trapping your hand forever.

Common sense tells us that this is highly implausible, but how can we prove for sure that it can't work? Just like the proof for the halting problem discussed in the box above, it works by contradiction. We assume that there is a solution, get it to work on itself, and find that this

leads to a paradoxical situation. That might sound perplexing, but it's simple. Just place your hand in the mouth of the statue and say, "I won't be able to remove my hand". Whatever happens, the oracle is doomed to failure. If the mouth snaps shut you'd spoken the truth, so it shouldn't have trapped your hand. If nothing happens you uttered a false statement, so it should have trapped your hand. Clearly the predicted behaviour is impossible, just like a solution to the halting problem.

THE HALTING PROBLEM

If you've ever written a computer program you'll know all about infinite loops. Assume that you've written a program that asks for a number – we'll call that the data – and displays its logarithm. But it's all too easy, in writing the program, to make a mistake that would cause it to loop forever rather than providing an answer and finishing. Alternatively, the program might give an answer and finish with some values of the data but loop forever with others.

The halting problem asks whether a given combination of program and data will ever halt (finish), or if it will loop forever. Originally it related to a Turing

machine (see page 222), but the same applies to an ordinary computer. An obvious way of finding out is just to try it. With a simple program such as our logarithm calculator, which ought to provide an answer in a fraction of a second, this would work well. But other programs might be designed to do something so complicated that they could run for hours, days or even months before halting. What's more, the time taken to come up with an answer might depend on the data.

Running the program to see if it will halt isn't an option. How long do you leave it running before you decide it isn't going to halt? Perhaps you'd leave it six months then terminate it manually because you'd decided it was in an infinite loop. But how could you be sure it wouldn't have come up with the goods and halted had you left it for just another clock cycle – another fraction of a second? It's impossible to tell whether a given combination of program and data will halt. If you want to know why, take a look at the proof above.

OTHER UNCOMPUTABLE PROBLEMS

Common sense tells us that a universal oracle is impossible, but it's quite surprising that such a logical and well-defined task as the halting problem can't be solved. Is this a single bizarre example, a unique curiosity, or are there other uncomputable problems? To find out, we spoke to Dr Mike Stannett, a theoretical computer scientist at



The Mouth of Truth in Rome claims to be a universal oracle, but is it a fake – and how do we prove it?



Sheffield University. "When someone wants to prove a problem is uncomputable, they do it by 'reducing it' back down to the halting problem," said Stannett. "They show that if their problem could be solved, so could the halting problem."

Our universal oracle illustrates Stannett's point. We've shown it's uncomputable, but even if we didn't know that, the fact that the halting problem is uncomputable implies that a universal oracle can't exist. If it did exist, we would only need to ask it if a given combination of program and data will halt; as we know, it's impossible to tell.

We asked Stannett for other examples of uncomputable problems. It turns out lots of problems relating to computer programs are uncomputable, as are others that seem totally unrelated to the halting problem. Stannett referred to a question that's of interest to anyone who travels by bus: 'Is the bus just late, in which case I should wait a bit longer, or has it been cancelled, in which case I should think of an alternative?'

By modelling the flow of traffic that affects the bus's progress, and by modelling the bus company's financial constraints that affect any decision to cancel a service, it should be possible to answer the question. However, as Stannett pointed out, "What the halting problem potentially tells us is that, even with all this information, it may not be possible to resolve the issue".

Stannett described a problem in mathematics, as opposed to logic, that can't be solved. "The best-known example of an uncomputable problem, apart from the halting problem, is Hilbert's Tenth Problem about Diophantine equations. There is no general algorithm which, when given a set of

polynomials with integer coefficients (such as $x^n + y^n + z^n = 0$), can decide whether they have an integer solution."

A couple of examples should clarify this for the non-mathematically minded. A simple example of a Diophantine equation is $x^3 - 27 = 0$ and it's easy to spot that the answer is three. But what of a more complicated example, such as $5x^5 - 17y^{12} + 244 = 0$? Mathematicians have found ways of solving some Diophantine equations, but in 1970 Yuri Matiyasevich proved that it is impossible to come up with a general method that can tell whether there's a whole number solution.

ENTER THE HYPERCOMPUTER

Uncomputable problems might foil an ordinary computer, but scientists are now talking about hypercomputers: those that can compute the uncomputable. A common type of hypercomputer is one that doubles in speed with every tick of its internal clock. If a 3GHz Pentium 4 could manage that trick, its first tick would take a three billionth of a second, its next tick would take a six billionth and so on. At first this seems odd. Surely if a computer could run at more than 3GHz it would do so from square one rather than ramping up with each tick? However, for a computer to be twice as fast it has to be twice as small. So if a computer could make a half-size replica of itself before passing on its program (including the instructions for self-replication) to that replica, we would have something that doubles its speed with each tick.

The astonishing thing about this computer is that it would be able to solve the halting problem. We know that the halting problem can't be solved

by analysing the program and its data, and we can't just try it out to see if it halted because we'd have to let it run forever before we knew for sure. However, a computer that doubles its speed every clock cycle would execute an infinite number of clock cycles in the period of just two of its initial clock cycles (because $1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} \dots$ never quite reaches two). So all you have to do is run the program on this computer and, if it's going to halt, it will do so within two of its initial clock ticks.

So is the ability to perform an infinite amount of computation in a finite time a feature of all hypercomputers? After all, most uncomputable problems seem to relate to the halting problem in some way. "Yes," said Apostolo Syropoulos, a theoretical computer scientist who has written a book on hypercomputation. He describes an even more bizarre hypercomputer. "It involves a pair of computers: one asks a problem and one solves the problem. The one that solves the problem operates in the normal universe, while the one that asks the problem enters an accelerated state to make use of the peculiarities of general and special relativity. Once there is a solution, the computer stops and comes back. Whether this is possible or not is an entirely different issue. All I can say is the scenario is feasible when the waiting computer enters the outer event horizon of a slowly rotating black hole."

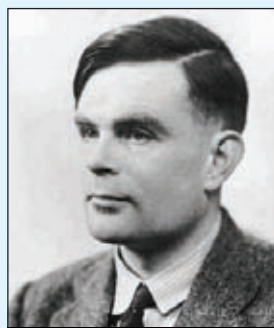
Syropoulos was most enthusiastic about the potential of analogue computers. The amount of work a digital computer can do in a clock cycle depends on the width of its registers. So a 32-bit computer can do twice as much work in a clock cycle as an otherwise-identical 16-bit computer, a 64-bit computer can do twice as much again and so

TURING: THE MAN AND THE MACHINE

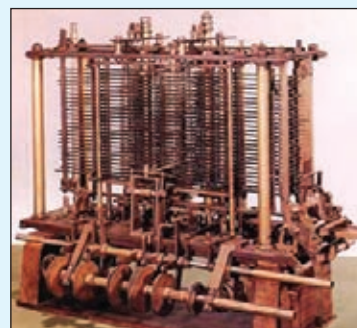
Imagine a machine that has an internal state (for example, a value like 0 or 1 or 2), which travels back and forth along an infinitely long paper tape, divided into segments along its length. Each segment can contain one symbol. When the machine lands on a segment it uses a look-up table to determine what to do depending on its internal state and the symbol on the tape. The action defined in the table involves writing a symbol on the tape in place of the existing symbol (although it could be the current symbol, in which case there is no change), adopting a new internal state and then moving one position to the left or right on the tape. As soon as the machine has moved, the process is repeated. This continues until the machine finds a combination of internal state and tape symbol that isn't in the look-up table. At this point, the computation is complete and the computer halts.

Now take a look at an example of how this strange computer can perform arithmetic. To start, the tape contains two numbers, one after the other, in unary representation. The unary representation of a number is simply the number of 1s, so if the initial data was a 2 and a 3 the tape would contain the pattern 01101110 and every other segment along its infinite length would contain a 0. Now let's assume that the machine starts off on the left-most 0 with the internal state 0 and that it has the following look-up table:

INPUT		ACTION		
State	Tape symbol	Change state to	Change symbol to	Move
0	0	0	0	Right
0	1	1	0	Right
1	0	2	1	Right
1	1	1	1	Right



Alan Turing devised the proof that the halting problem is uncomputable



The Turing Machine, which formed the basis of modern, conceptual computing

It's a fairly simple paper exercise to discover that the machine will halt with the following pattern on the tape: 0111110, the unary representation of 5. In general, you'll discover that its function is to add two numbers together.

This conceptual computer is called a Turing Machine, named after computer pioneer Alan Turing. Different Turing Machines perform different tasks, but Turing proved that it was possible to build a Universal Turing Machine that could behave like any other Turing Machine and thereby perform a whole range of different tasks. It achieved this feat by first reading from the tape a definition of the Turing Machine that it was going to emulate. This is an obvious analogy to a modern computer program, and the similarity doesn't end there. Much of the theory of computing relied on this conceptual model of a computer.

Turing went on to say that any problem that is computable can be solved on a Universal Turing Machine. An extension to this is that today's computers are no more powerful, fundamentally, than a Turing Machine; if a problem is uncomputable on a Turing Machine, it's uncomputable on today's fastest supercomputer. In fact, the proof that the halting problem is uncomputable was first devised by Turing by considering a Turing Machine.



on. But there's a limit to any conventional digital computer's register width and hence a limit to the amount of work it can do in any period of time. In theory, an analogue computer works on values that are continuously variable, which means they can store values to infinite precision and so do an infinite amount of work in a finite time. Practical considerations mean that no analogue computer to date has accomplished this rather neat trick.

Being able to do an infinite amount of work in a finite time may be a property of a hypercomputer, but it's not a definition of one. So what do we mean by a hypercomputer? Is it simply a computer that can solve the halting problem and, by implication, all the other problems that are related to it? According to Syropoulos, there is no commonly accepted definition. Many people believe anything that claims to be a hypercomputer must be able to solve the halting problem, while others have a more relaxed definition.

Stannett referred to levels of uncomputability, and hence the possibility that a computer that could solve the halting problem but be stumped by even more uncomputable problems. As he pointed out: "Even if we developed new technology that could solve the halting problem for Turing machines, it almost certainly wouldn't be able to solve its own halting problem."

FACT OR FICTION?

If you like to think that you have a firm grasp on reality, all this talk of continually shrinking computers and machines at the outer event horizon of a slowly rotating black hole will be greeted with a large dose of scepticism. Surely both these ideas, and others such as quantum computers with an infinite super-position of states, are totally impractical? Consider the shrinking computer, otherwise known as the Zeno machine. With clever engineering, scientists might be able to produce a computer that could build a half-sized replica of itself and pass its programming on to that replica. But before too long, it would approach atomic dimensions, at which point the shrinking would have to stop. All the other hypercomputing models seem to be scuppered by the laws of physics, too.

The point is they are all 'thought computers', ideas dreamed up by theoreticians to test out theories of computability. But Syropoulos wouldn't let us get away with discounting them on these grounds. "Even the Turing machine is a notational device," he pointed out. Yet as we all know, it paved the way to today's PC. And in anticipation of our next question, he explained how a Zeno machine might be possible after all. "It is not that a Zeno machine is unfeasible, it is that it depends on the 'fact' that space and time are continuous and not discrete as expected by quantum gravity. However, it is quite interesting that recently two American physicists provided evidence that space and time are indeed continuous."

So will any of these potential techniques lead to a practical hypercomputer? Syropoulos gave an unequivocal yes. "The optical model of computation and in general the analogue devices that have hyper-computational characteristics may lead one day to the construction of real hypermachines," he suggested. "Whether this day will be tomorrow or after 100 years, I really do not know." So we should probably not get too excited about the prospect of solving Hilbert's Tenth just yet, then.

CREATING RANDOM NUMBERS

Type `=RAND()` into a cell of an Excel worksheet and you'll find that a random number between zero and one appears. Type the same formula into another cell and a different random number will appear. So it might sound strange for us to suggest that computers can't produce random numbers and that their generation is another uncomputable problem.

Random numbers produced by Excel, or any computer program, are what mathematicians call pseudo-random. They seem to be random but, because a computer program is a fixed set of instructions, those apparently random numbers actually follow a mathematical sequence, albeit a convoluted one. For many applications pseudo-random numbers are good enough. But bearing in mind that random numbers are used in coding secret messages and in lotteries, we can see that numbers that follow a sequence just won't do. If someone manages to analyse the sequence, it might be possible to predict the next number.

Generating truly random numbers doesn't need a hypercomputer but it needs something other than an ordinary digital computer. The National Lottery uses a machine with balls that are tossed randomly to and fro before being selected. This might work when there's a handful of draws each week, but when lots of random numbers are needed a different solution is required.

In 2004 the government-run National Savings and Investments introduced the fourth generation of Electronic Random Number Indicator Equipment (ERNIE), used to generate winning numbers for the monthly Premium Bond prize draw. Its task is to produce around two million random 11-bit numbers each month. It may look like a computer and it might produce its results digitally, but it uses a special chip: the Intel Random Number Generator (RNG). This chip relies on thermal noise to make random numbers.



Electronic Random Number Indicator Equipment, otherwise known as ERNIE, produces around two million random 11-bit numbers each month

Intel's random number generator has, at its heart, a pair of resistors. Like all electrical resistors, the random movement of electrons causes random currents in these resistors. But the current in the resistors also depends on non-random local conditions such as the magnetic field emitted by nearby circuitry. This is why two resistors are used. Because they're so close together, any local effects will be almost identical in the two resistors yet the random effects will be different. By subtracting the output of one resistor from that of the other, the local effects are cancelled out. The random electric signal (which is analogue) is fed into a voltage-controlled oscillator that produces a square wave digital output. Because this signal is derived from the random output of the resistors, the transitions from zero to one and vice versa occur randomly.

The final piece of the jigsaw is to use the transitions of this oscillator to sample the output of a higher frequency oscillator. If a transition of the random oscillator occurs when the high-frequency oscillator is in its 'zero' state, a zero forms part of the chip's random output; if it occurs when the oscillator is in its 'one' state, a one results.

The Intel RNG

Producing genuinely random numbers requires something beyond the capabilities of an ordinary digital computer, but the Intel Random Number Generator may have come up with the solution

